VLSI - Very Large Scale Integration

Combinatorial Decision Making and Optimization

Module 1

Eric Rossetto

Matricola: 982594

eric.rossetto@studio.unibo.it

GitHub

Xiaowei Wen

Matricola: 982501

xiaowei.wen@studio.unibo.it

GitHub

Academic year 2020-2021

Abstract

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. A typical example is the smartphone. The modern trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, has pushed the integration of more and more functions of cellphone circuitry into a single silicon die (i.e. plate). This enabled the modern cellphone to mature into a powerful tool that shrank from the size of a large brick-sized unit to a device small enough to comfortably carry in a pocket or purse, with a video camera, touchscreen, and other advanced features.

Contents

List of Tables

1 Introduction

1.1 What?

This is a project about combinatorial problems, the main topics done are:

- Modelling of a combinatorial problem;
- Minizinc as a CSP;
- SAT/SMT;

1.2 How?

At the beginning of the project, we reasoned over the problem, thinking about how to present the it in a model, with which variables and constraints.
Then, we tried to improve the model by adding symmetry breaking constraints. And lastly, as asked by the project, we considered how to modify the model if we want to consider also the rotation of the circuits.
When we started working, we created different script in python:

- a converter from *.txt* in *.dzn*, in a format which is compatible with the model we thought;
- a solution checker that checks if a certain solution, produced by the model, if a certain solution is correct, that mean if circuits are not over-lapped;
- a solution to picture converter, that produce an image representing the pieces of circuits with different colour given a solution.
- a script which executes a Minizinc model with and saves the time of execution;

Once we got everything we need, we started coding models in Minizinc, and when we were sure that every model works, we translated the model in the SMT in a python notebook. Then we did many tests with Minizinc, such as using different solvers, different search strategies, and different optimization levels.

1.3 How to get the same results

For Minizinc models, you should use the script we created (to do so, you need to add Minizinc Command line tool into the environment variable *PATH*), because in the script we set some parameters like solver to use, timelimit, randomseed. Otherwise you can use the solver configuration file, but notice that solving an instance in the Minizinc IDE is evidently slower than solving the same instance by using the script, this difference is caused by the Minizinc Command line tools.

For the SMT, we can just run the model.

We executed our model on a *Surface book 2* with processor Intel i7-8650U and 8 GB of Ram. If you run our models in other machine, you may obtain different results.

We consider a solution as such when the solver terminates within 300 seconds, meanwhile those instances which execution are aborted because of the time-out, and given a solution which is the optimal, are considered as non solved.

2 Models

2.1 Base Model

The model chosen simply describes the circuits by means of rectangles. The silicon plate is naturally represented by a rectangle of a certain width, let it be *WIDTH*, and a certain height, call it *HEIGHT*. In addition, we are given a set of N rectangles, ideally representing our circuits, each of a predefined width w_i and height h_i. Each rectangle is handily represented by a tuple (x_i, y_i) with $x_i \in 0, \ldots,$ *WIDTH* and $y_i \in 0, \ldots,$ *HEIGHT*, namely the coordinates of the left-bottom corner of the considered circuit. The objective is to find the smallest enclosing box, the silicon plate, that contains these rectangles without any overlapping between them or in other words find a configuration of coordinates making the enclosing rectangle's area minimal. In this specific project the *WIDTH* of the box is passed as a parameter, and as such is fixed, therefore the objective simplifies to a minimization of the height of the silicon plate.

2.1.1 Without rotation

The first formulation does not allow any rotation of the circuits, then each circuit must be placed in a fixed orientation with respect to the others. We will define the constraints to be satisfied by means of logical propositions. First of all we have to explicit the constraints that define boundaries in which the rectangles must be placed. Given a generic circuit c_i, positioned in (x_i, y_i) and a set of indexes $S = \{1, \ldots, N\}$; we could formulate the *boundary constraints* as follows:

$$\forall i \in S.(x_i + w_i \leq \mathit{WIDTH}), \tag{1}$$

$$\forall i \in S.(y_i + h_i \leq \mathit{HEIGHT}). \tag{2}$$

These constraints ensure that the circuits stays in the plate. In addition to these constraints we have to define the *non-overlapping constraint*. Given two circuits c_i, positioned

in (x_i, y_i), and c_j, positioned in (x_j, y_j), each one with its height and width:

$$\forall i, j \in S \text{ where } i \neq j$$

$$x_i + w_i \leq x_j \vee \tag{3}$$

$$x_i - w_j \geq x_j \vee \tag{4}$$

$$y_i + h_i \leq y_j \vee \tag{5}$$

$$y_i - h_j \geq y_j. \tag{6}$$

Two circuits are not overlapping if one of these inequalities is true, in particular we can say that:

- if (3) is true then c_i is on the left of c_j,
- if (4) is true then c_i is on the right of c_j,
- if (5) is true then c_i is above c_j,
- if (6) is true then c_i is below c_j.

The last ingredient of our model that we have to properly define is the concept of *HEIGHT*. Indeed, we are searching for solutions that aim to minimize it so it comes natural to come up with the function *height* that computes *HEIGHT*:

$$height() = \max_i (y_i + h_i)$$

Note that, *HEIGHT* can range freely between 0 and the *maximum allowable height* which is the result of the following computation:

$$max\_height() = \sum_i h_i.$$

Therefore we can introduce the *domain constraints* to the *HEIGHT* variable.

$$HEIGHT \geq 0 \tag{7}$$

$$HEIGHT \leq max\_height(). \tag{8}$$

This could be furthermore improved by reasoning on the fact that the resulting enclosing plate should be at least high as the shortest circuit given. This can be useful to reduce the

search space. Said that we can introduce the *minimum allowable height* that it is computed as $min\_height() = \min_i h_i$ and then modify accordingly the *domain constraint*(7):

$$HEIGHT \geq min\_height() \tag{9}$$

In the end, the problem boils down finding the coordinates (x_i, y_i) where $i \in 1, \ldots, N$ such that

$$\text{minimize } height().$$

2.1.2 With rotation

The second formulation takes into account rotations. We can see that a rectangle shows an unique different configuration only if it is rotated by 90°. It can be easily seen that this particular rotation simply switch w and h; this can be easily achieved by introducing a new variable $rot_i \in \{0, 1\}$ for each circuit i. The additional variable is used to indicate the change of orientation of the rectangle:

- $rot_i = 0$ - circuit i is in its original position,
- $rot_i = 1$ - circuit i is rotated by 90°.

Said that we can modify the constraints mentioned in the previous section accordingly to the addition of the new variable. The *boundary constraints* can be substituted by:

$$\forall i \in S.(x_i + rot_i h_i + (1 - rot_i)w_i \leq WIDTH), \tag{10}$$

$$\forall i \in S.(y_i + rot_i w_i + (1 - rot_i)h_i \leq HEIGHT), \tag{11}$$

while the *non-overlapping* ones in this way:

$$\forall i, j \in S \text{ where } i \neq j \tag{12}$$

$$x_i + rot_i h_i + (1 - rot_i)w_i \leq x_j \vee \tag{13}$$

$$x_i - rot_j h_j - (1 - rot_j)w_j \geq x_j \vee \tag{14}$$

$$y_i + rot_i w_i + (1 - rot_i)h_j \leq y_j \vee \tag{15}$$

$$y_i - rot_j w_j - (1 - rot_j)h_j \geq y_j. \tag{16}$$

The height function should take into account rotations too, as well as the *maximum allowable height* because now a rotation could modify substantially the maximum height

of the silicon plate. Thus:

$$height() = \max_i (y_i + rot_i w_i + (1 - rot_i) h_i) \qquad max\_height() = \sum_i \max_i (w_i, h_i).$$

Other constraints or declarations remain the same.

3 Constraint Satisfaction Programming - CSP

In the context of Artificial Intelligence, a *constraint satisfaction* is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution to a such a program is a set of values to be assigned to the variables that satisfies all the constraints defined. As *modeling language* we have chosen **MiniZinc** to express variables and constraints of the mentioned problem.

3.1 Minizinc Implementation

The MiniZinc implementation follows closely the logical formulation presented in §2.

Input formatting Each instance of VLSI is presented as a .dzn file. The first line denotes the width of the silicon plate `width`, the second line the number of circuits to place `n_circuits` and lastly there is a bi-dimensional array `dims` that defines the dimensions of each circuit.

3.2 Without rotation

To implement the model described in §2.1.1 we propose the following variable organization:

- **Parameters**, which is the data given in input, then expressed in the MiniZinc language. In this section, we have defined the `max_height` and `min_height` which are the result of the aforementioned functions.
- **Decision Variables**, we have two decision variables, namely the variables that we would like to assign a value:
  - the coordinates of each rectangle are gathered in an array `corner_coords` where for a generic row c, `corner_coords[c, 1]` denotes the horizontal coordinate and `corner_coords[c, 2]` denotes the vertical coordinate.
  - the height of the silicon plate, `height`.

3.2.1 Simplest model

The only auxiliary function here is the one that computes the height and the corresponding constraints are listed here.

```
% No-overlap constraint
constraint forall(i,j in CIRCUITS where i!=j)(
  corner_coords[i,1] + dims[i,1] <= corner_coords[j,1] \/
  corner_coords[i,1] - dims[j,1] >= corner_coords[j,1] \/
  corner_coords[i,2] + dims[i,2] <= corner_coords[j,2] \/
  corner_coords[i,2] - dims[j,2] >= corner_coords[j,2]
);

% x, y of each block should have as starting coordinate (0,0)
constraint forall(i in CIRCUITS)(corner_coords[i, 1] >= 0);
constraint forall(i in CIRCUITS)(corner_coords[i, 2] >= 0);

% Boundaries constraint
constraint forall(i in CIRCUITS)(corner_coords[i, 1] +
  dims[i, 1] <= width);
constraint forall(i in CIRCUITS)(corner_coords[i, 2] +
  dims[i, 2] <= height);

% Height domain constraint
constraint height >= min_height /\ height <= max_height;
```

20

3.2.2 With global constraints

In order to improve the performance of the previous model we have to introduce some *global constraints* in the code. The most obvious above all is `diffn` which constrains each rectangle to be non-overlapping given its origin and sizes. First of all we have to remove the *no-overlap* constraint from the previous model and add the following:

```
constraint diffn([corner_coords[i, 1] | i in CIRCUITS],
  [corner_coords[i, 2] | i in CIRCUITS], [dims[i, 1] | i in
  CIRCUITS], [dims[i, 2] | i in CIRCUITS]);
```

Another substantial improvement could be obtained by adding the `cumulative` for each one of the axes, specifically we are requiring that a set of tasks given by start times **s**(namely the array of x_i's), durations **d**(namely the array of the horizontal sizes), and resource requirements **r**(namely the array of the vertical sizes), never require more than a global resource bound **b**(namely *height*) at any one time. A similar reasoning can be done for the other axes. Said that we can simply add below the `diffn` constraint the following:

```
constraint cumulative([corner_coords[i, 1] | i in CIRCUITS],
  [dims[i, 1] | i in CIRCUITS], [dims[i, 2] | i in CIRCUITS],
  height);
constraint cumulative([corner_coords[i, 2] | i in CIRCUITS],
  [dims[i, 2] | i in CIRCUITS], [dims[i, 1] | i in CIRCUITS],
  width);
```

3.3 With rotation

As we saw in §2.1.2, we have to introduce in the MiniZinc model a new uni-dimensional array `rot` that tells us if a generic circuit is rotated by 90° or not. Therefore the constraints of the model proposed in §3.2.1 can be easily modified in such a way:

```
% No-overlap constraint
constraint forall(i,j in CIRCUITS where i != j)(
  corner_coords[i,1] + rot[i] * dims[i,2] + (1 - rot[i]) *
   dims[i,1] <= corner_coords[j,1] \/
  corner_coords[i,1] - rot[j] * dims[j,2] - (1 - rot[j]) *
   dims[j,1] >= corner_coords[j,1] \/
  corner_coords[i,2] + rot[i] * dims[i,1] + (1 - rot[i]) *
   dims[i,2] <= corner_coords[j,2] \/
  corner_coords[i,2] - rot[j] * dims[j,1] - (1 - rot[j]) *
   dims[j,2] >= corner_coords[j,2]
```

```
);

% Boundaries constraint
constraint forall(i in CIRCUITS)(corner_coords[i, 1] + rot[i]
  * dims[i,2] + (1 - rot[i]) * dims[i,1] <= width);
constraint forall(i in CIRCUITS)(corner_coords[i, 2] + rot[i]
  * dims[i,2] + (1 - rot[i]) * dims[i,1] <= height);

% Height domain
constraint height >= min_height /\ height <= max_height;

% Rot domain
constraint forall(i in CIRCUITS)(rot[i] >= 0 /\ rot[i] <= 1);
```

3.3.1 With global constraints

What we have done in §3.2.2 is basically the same here, the only thing is that we have to properly consider the possibility of rotation and so the fact that in `diffn` the sizes can change.

```
constraint diffn([corner_coords[i, 1] | i in CIRCUITS],
  [corner_coords[i, 2] | i in CIRCUITS], [(1 - rot[i]) *
  dims[i, 1] | i in CIRCUITS], [rot[i] * dims[i, 2] | i in
  CIRCUITS]);

constraint cumulative([corner_coords[i, 1] | i in CIRCUITS],
  [dims[i, 1] | i in CIRCUITS], [dims[i, 2] | i in CIRCUITS],
  height);
constraint cumulative([corner_coords[i, 2] | i in CIRCUITS],
  [dims[i, 2] | i in CIRCUITS], [dims[i, 1] | i in CIRCUITS],
  width);
```

3.4 Symmetry breaking

Symmetry is an issue with the problem we are considering: for each solution there exists other seven symmetric variants: rotation by 90°, 180° and 270° and their respective vertically flipped versions (see figure 1).

We could reduce the solver's work by reducing the number of solutions, thus by keeping only the non-symmetric ones. This can be done by placing the circuit with the largest area in the left-hand quarter of the silicon plate, namely by saying that $x_{big} \leq \frac{WIDTH}{2}$ and $y_{big} \leq \frac{HEIGHT}{2}$. This could be easily done in MiniZinc by adding this particular constraint:

```
constraint symmetry_breaking_constraint(
    % This breaks the symmetries on the vertical axis
    corner_coords[biggest(), 1] * 2 <= width
    % This breaks the symmetries on the horizontal axis
    /\ corner_coords[biggest(), 2] * 2 <= height
    );
```

By looking at the figure 1, after the evaluation of the *symmetry breaking constraint* the only solutions kept are 2 over the original 4, in the case we are not allowing rotation, whilst 4 over 8, in case of rotation.

A stronger *symmetry breaking constraint* will place the biggest circuit in the lower-left corner of the silicon plate, namely in $(0, 0)$:

$$x_{big} == 0 \wedge y_{big} == 0.$$

Nevertheless this constraint would not likely produce the desired effects, often slowing the solving process done by the MiniZinc's solver and as such it will not be considered in the CSP models comparison. The relative implementation is pretty straightforward:

```
constraint symmetry_breaking_constraint(
    corner_coords[biggest(), 1] = 0
    /\ corner_coords[biggest(), 2] = 0
    );
```

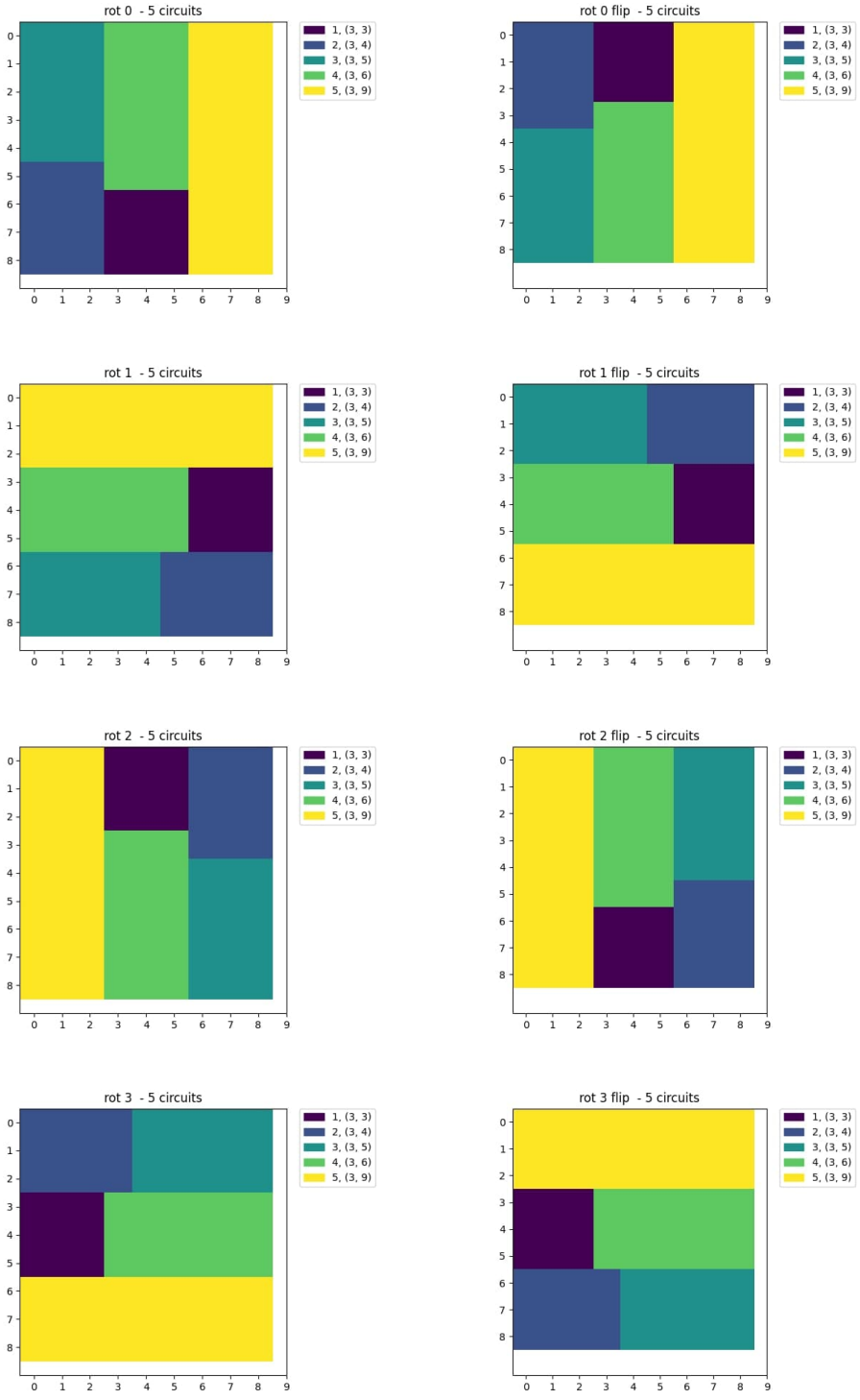

Figure 1: Second instance - all possible symmetric variants of a solution. Rot X indicates that the plate has undergone a rotation of X*90 degrees.

3.5 MiniZinc models comparisons

The different models presented so far are now compared using a predefined configuration of the solver in order to avoid randomness in the results.

Model	Rotation	Symmetry breaking	Solved instance	Mean time / instance (s)	Total time* (s)
1.0.0	No	No	23	30.79	708.20
1.3.0	No	No	25	6.86	171.56
1.0.0	Yes	No	14	25.15	352.07
1.3.0	Yes	No	17	43.59	741.02
1.0.0	No	Yes	14	35.37	495.19
1.3.0	No	Yes	22	21.19	466.13
1.0.0	Yes	Yes	21	26.32	552.82
1.3.0	Yes	Yes	26	17.48	454.53

Table 1: MiniZinc models performances. Model 1.0.0 is the model without any global constraint, model 1.3.0 is the one with `diffn` and `cumulative`. (*) the total time is computed by summing up successfully solved times.

The best model is the *1.3.0* the one with rotation and symmetry breaking, we think this is caused by the optimization done by the solver when we are using global constraints implemented in Minizinc. For further details, you can consult the file pdf **CSP_performance.pdf**

4 Satisfiability modulo theories - SMT

In computer science and mathematical logic, the **SMT** problem is a decision problem for logical formulas with respect to combinations of background theories expressed in **FOL**. SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalized approach to constraint programming.

4.1 SMT implementation

To implement models in **SMT** we will use the library **Z3** in python notebooks.

4.1.1 Without rotation

To implement the model described in model without rotation we used as variables:

- **x** to represent the X coordinates of the circuit;
- **y** to represent the Y coordinates of the circuit;
- **Height** to represent the maximum height reached by the circuits;

And as parameters we have:

- N which represent the number of circuit;
- S to represent the set of circuit from 0...N;
- w_i to represent the width of circuit i;
- h_i to represent the height of circuit i;
- **WIDTH** which represent the width of the plate;

Then we added to the solver the following conditions:

$$width\_boundaries \equiv \forall i \in S.(x_i \geq 0 \wedge x_i + w_i \leq WIDTH), \tag{17}$$

$$height\_boundaries \equiv \forall i \in S.(y_i \geq 0 \wedge y_i + h_i \leq HEIGHT), \tag{18}$$

$$no\_overlap\_cons \equiv \forall(i, j \in S \text{ where } i \neq j).(\tag{19}$$

$$(x_i + w_i \leq x_j) \vee \tag{20}$$

$$(x_i - w_j \geq x_j) \vee \tag{21}$$

$$(y_i + h_i \leq y_j) \vee \tag{22}$$

$$(y_i - h_j \geq y_j)). \tag{23}$$

Hence, as the conjunction of all constraints we'll have:

$$all\_constraints \equiv width\_boundaries \wedge height\_boundaries \wedge no\_overlap\_cons \tag{24}$$

Finally, by minimizing the **HEIGHT** we are reducing the empty area created by placing the circuits in the plate as the sum of area of the circuit is constant, although the **WIDTH** is fixed.

4.1.2 With rotation

For the version of the model considering also the rotation, based on the previous model, we added a list of binary variables, which domain is defined as:

$$rot\_domain \equiv \forall i \in S.(rot_i \geq 0 \wedge rot_i \leq 1) \tag{25}$$

And we are considering the fact that:

- rot_i is equal to 0, then the circuit i is not rotated;
- rot_i is equal to 1, then circuit i is rotated by 90°;

Then, we slightly modified constraints of the no rotation version as follows:

$$width\_boundaries \equiv \forall i \in S.(x_i \geq 0 \wedge x_i + (1 - rot_i)w_i + rot_i h_i \leq WIDTH) \tag{26}$$

$$height\_boundaries \equiv \forall i \in S.(y_i \geq 0 \wedge y_i + (1 - rot_i)h_i + rot_i w_i \leq HEIGHT) \tag{27}$$

We simply considered that if a certain circuit i is rotated, the w_i and h_i will be exchanged, so to compute the coordinates of the opposite corner, we will have to do the same.

$$no\_overlap\_cons \equiv \forall(i, j \in S \text{ where } i \neq j).(\tag{28}$$

$$(x_i + rot_i * h_i + (1 - rot_i) * w_i <= x_j) \tag{29}$$

$$\vee\, (y_i + rot_i * w_i + (1 - rot_i) * h_i <= y_j) \tag{30}$$

$$\vee\, (x_i - rot_j * h_j - (1 - rot_j) * w_j >= x_j) \tag{31}$$

$$\vee\, (y_i - rot_j * w_j - (1 - rot_j) * h_j >= y_j) \tag{32}$$

Also in this model, we set a objective function, the *height*, and our aim is to minimize it.

4.2 Symmetry breaking

There are two symmetry breaking conditions, the first one is composed by the following:

$$x_{big} * 2 \leq width \wedge y_{big} * 2 \leq height \tag{33}$$

In this case, we are forcing the biggest circuit to be placed in the a quadrant of the plate:

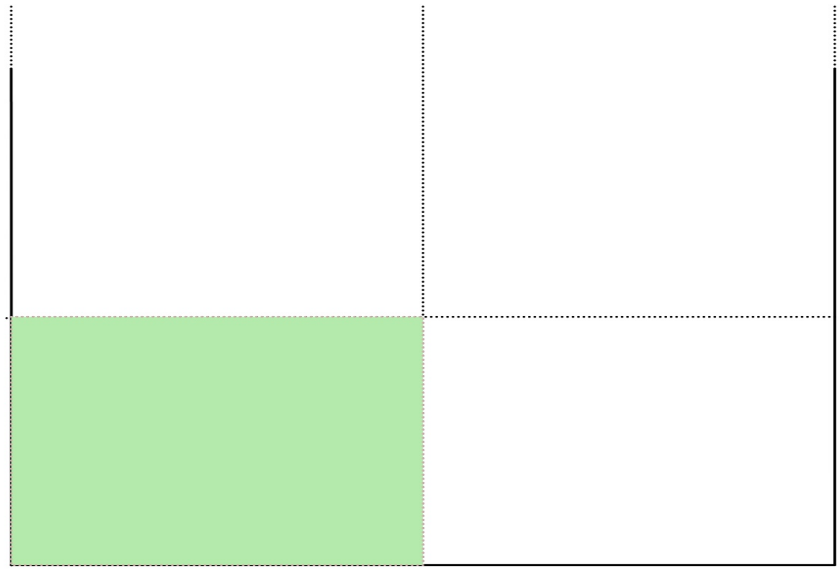

Figure 2: Green area is the possible position of the biggest circuit

Meanwhile, the second one is the extreme case of the previous one, because we are imposing these conditions:

$$x_{big} == 0 \wedge y_{big} == 0 \tag{34}$$

where *biggest* is the index of the circuit with the biggest area. And in this case, the biggest circuit will be placed in the lower-left corner of the plate. Differently from what has been done with the MiniZinc implementation, here both of the symmetry constraints will be kept since the extreme case seems to perform better overall.

4.3 SMT models comparisons

Now, we are going show the performance of different models over the 40 instances:

Model	Rotation	Symmetry breaking	Solved instance	Mean time(solved instances) (s)	total time (s)
A	No	No	22	16.78	369.11
B	No	Yes	24	29.51	708.27
C	Yes	No	14	38.94	545.23
D	Yes	Yes	14	60.00	840.03

Table 2: SMT model performances

We can notice from the table that the model B has solved two instances in more w.r.t. the model A (instance 19 and 28 respectively with 242.95s and 180.94s), and because of this, it's mean time is higher than the one of A.

4.3.1 No rotation vs rotation

From the previous table, we can notice a clearly improvement of the performance between model without rotation and the one with rotation, this is because in the latter one we have to consider when pieces are rotated, and the combination is 2^N times in more w.r.t. first one.

4.3.2 Two cases of symmetry breaking

As we said in the §4.2 there are two cases of symmetry breaking, the following table shows the different of performance when executing the models for instances from 10 to 19 as they are those more difficult to solve. We will only consider the model with rotation.
For the instances which are not solved within 300s, we abort the execution and consider the execution time as 0s.
The extreme case can solve 2 more instances w.r.t. the normal one, and if we recompute the mean time for extreme case, considering only instances solved by the normal case, the result would be *83.19*, which is much lower than the normal case.

	Normal case	Extreme Case
Instances 10	149.46	108.86
Instances 11	0.00	0.00
Instances 12	6.18	29.22
Instances 13	151.64	101.55
Instances 14	0.00	162.97
Instances 15	17.47	130.00
Instances 16	0.00	0.00
Instances 17	227.20	46.30
Instances 18	0.00	250.02
Instances 19	0.00	0.00
Solved instances	5	7
Mean (All instances)	205.1956	172.8912
Mean (Solved instances)	110.40	118.42
Total time	2051.956	1728.912

Table 3: Execution statistics of instances 10 to 19.

What we can see from the graph is that the extreme case works better.

Differently from the **CSP**, we can see a improvement of the performance of the extreme case of symmetry breaking.

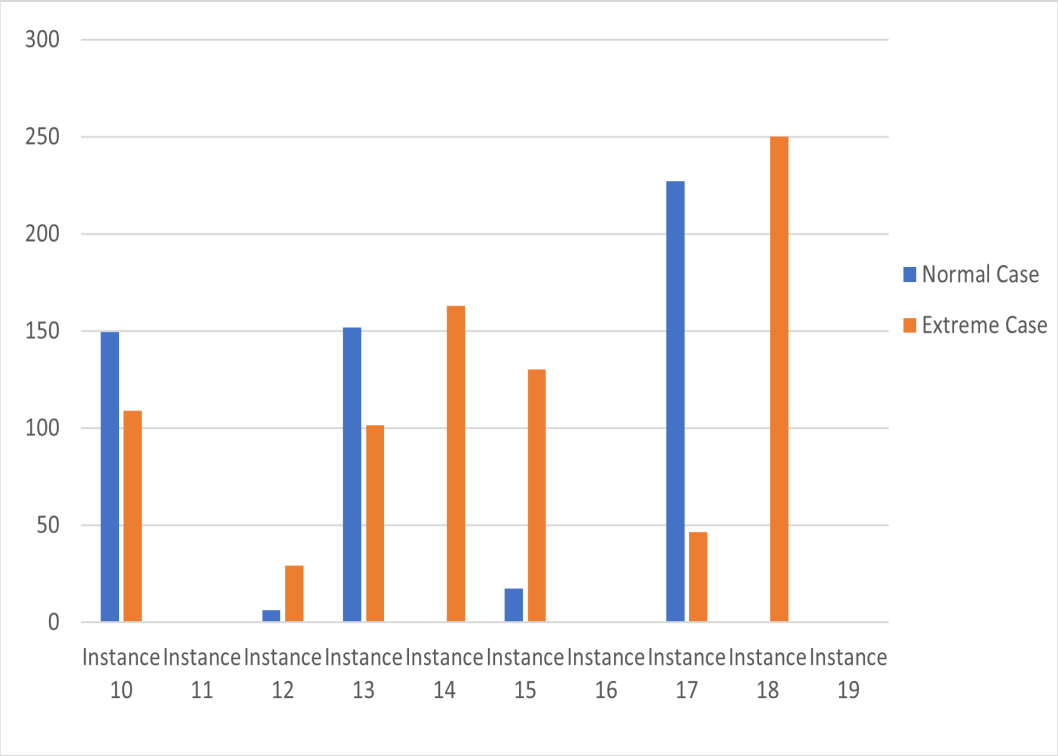

Figure 3: Symmetry breaking comparison.

5 Results analysis and comparison

In the following tables, we are going to compare the different versions of the model solving the instances from 10 to 19. When the instance is not successfully solved within 300 seconds, we set the time to *0.00*. Compared versions are:

- *CSP 1.0.0*: the base version of the model with only strictly needed constraints;
- *CSP 1.3.0*: the version of the model with global constraints *diffn* and *cumulative*;
- *SMT*: the same as CSP 1.0.0 but implemented in SMT.

5.1 CSP vs SMT

Instances	CSP 1.0.0	CSP 1.3.0	SMT
Instance 10	0.36	0.61	0.55
Instance 11	0.00	55.84	0.00
Instance 12	0.98	0.63	2.68
Instance 13	0.89	0.70	2.36
Instance 14	3.01	0.94	7.61
Instance 15	0.45	0.76	3.79
Instance 16	0.00	0.00	0.00
Instance 17	11.80	1.26	32.52
Instance 18	0.85	1.11	16.85
Instance 19	0.00	0.00	0.00
Solved instances	7	8	7
Average Solved(s)	2.62	7.73	9.48

Table 4: CSP VS SMT base

In this comparison, we can notice that **CSP 1.3.0** works better than others, because it can solve the instance 11, although it takes *55.84* seconds. If we recompute the average solved time not considering instance 11, we would obtain as mean time *0.75* second per instance.

5.2 CSP vs SMT with symmetry breaking

Instances	CSP 1.0.0	CSP 1.3.0	SMT
Instance 10	1.05	0.41	0.47
Instance 11	0.00	42.41	0.00
Instance 12	1.11	0.48	1.54
Instance 13	1.09	0.52	1.55
Instance 14	1.59	0.74	2.49
Instance 15	1.97	0.79	1.97
Instance 16	0.00	0.00	0.00
Instance 17	7.52	1.11	37.29
Instance 18	99.53	2.12	11.34
Instance 19	0.00	0.00	0.00
Solved instances	7	8	7
Average Solved(s)	16.27	6.07	8.09

Table 5: CSP VS SMT with symmetry breaking

In this version of the model, we can notice that **CSP 1.3.0** can solve the *instance 11* which is not solved by other models. For other instances, we can see that **CSP 1.3.0** can solve them in quite similar time w.r.t. other models.

5.3 CSP vs SMT with rotation

In the models with rotation, we can expect that the solving time are longer than the model without rotations, because we need to consider more variables. So, with the time-out of 300 seconds, we can solve less instances. Also in this comparison, we can see that the **CSP 1.3.0** works better, still, this could be caused by the optimizations done by the solver with global constraints.

Instances	1.0.0	1.3.0	SMT
Instance 10	26.05	0.50	206.93
Instance 11	0.00	150.08	0.00
Instance 12	0.00	1.44	4.55
Instance 13	19.52	0.88	0.00
Instance 14	264.99	72.59	0.00
Instance 15	1.05	32.46	36.76
Instance 16	0.00	0.00	0.00
Instance 17	0.00	187.57	0.00
Instance 18	0.00	0.00	61.27
Instance 19	0.00	0.00	0.00
Solved instances	4		
Average Solved	77.90	63.65	77.38

Table 6: CSP VS SMT with rotations

Instances	1.0.0	1.3.0	SMT
Instance 10	61.74	0.44	121.9323
Instance 11	0.00	133.78	0
Instance 12	135.53	0.81	154.3911
Instance 13	13.36	0.65	279.6322
Instance 14	0.00	1.16	212.5075
Instance 15	31.92	0.78	61.53585
Instance 16	0.00	0.00	0
Instance 17	0.00	2.20	0
Instance 18	0.00	5.74	0
Instance 19	0.00	0.00	0
Solved instances	4	8	5
Average Solved	60.64	18.19	166.00

Table 7: CSP VS SMT with rotation and symmetry breaking

5.4 CSP vs SMT with symmetry breaking and rotation

By combining the previous two cases, we can produce a model which is able to solve *8* instances, with average time *6.57*s. But still, other two models work worse than the **CSP 1.3.0**, this can be caused by the use of the global constraints.
Considering also the fact that the computational power required to satisfy symmetry breaking constraints and rotations is much higher.

6 Conclusion and future developments

6.1 Possible future developments

One of possible future development is to introduce new symmetry breaking as follows: once we have found a solution, we can represent it as a bi-dimensional matrix M, in which every value $M_{i,j} \in \{0, \ldots, N\}$, then it is possible to encode each circuit as a sub-matrix of M, such that for a generic circuit v, $M_{x_v \ldots w_v, y_v \ldots h_v} = v$. A possible example, representing the solution of figure 4, is the following:

$$\begin{bmatrix} 1 & 1 & 1 & 3 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 3 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 & 2 & 2 & 2 \end{bmatrix}$$

In order to reduce the number of symmetries to be computed by the solver we can impose an ordering between the matrix M and other possible permutation of it. In particular we can state that:

- to remove the original version flipped vertically:

  lex_lesseq(array1d(M), $M[i,j]$ where $i \in \{HEIGHT, \ldots, 1\}, j \in \{1, \ldots, WIDTH\}$);

- to remove the rotated version by 180°:

  lex_lesseq(array1d(M), $M[i,j]$ where $i \in \{HEIGHT, \ldots, 1\}, j \in \{WIDTH, \ldots, 1\}$);

- to remove the 180 degrees rotated and flipped vertically version:

  lex_lesseq(array1d(M), $M[i,j]$ where $i \in \{1, \ldots, HEIGHT\}, j \in \{WIDTH, \ldots, 1\}$).

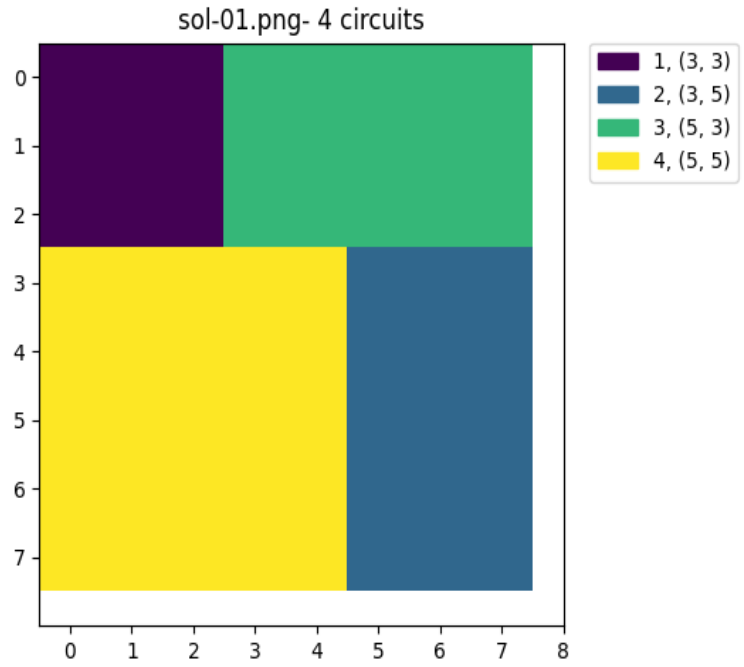

Figure 4: Solution as image

6.2 Final considerations

By developing this project, many theoretical concepts are clarified and applied in a practical way.

As first step, we reasoned on how to model a problem identifying which are parameters and variables, how to encode it in different languages, such as Minizinc and SMT, then we tried to improve the model by introducing in Minizinc global constraints, implied constraints or conditions, and finally symmetry breaking constraints. We though that this procedure is much efficient.

Then, we did different experiments by changing configuration parameters, for Minizinc models, we found that the fastest way to find the correct solution is to apply *first_fail* as the criteria to choose the variable, and *indomain_min* to choose the value.

We can say that **CSP 1.3.0** seems to perform better than **SMT** and **CSP 1.0.0**, at least it should asymptotically do so. Below we have summarized some datum about the model mentioned on the **40** instances given as reference:

- **Solved instances**: *26*;

- **Average solving time**: *17.48s*;
- **Total execution time for the solved instances**: *454.53s*

References

[1] Suphachai Sutanthavibul, Eugene Shragowitz, and J. Ben Rosen. "An analytical approach to floorplan design and optimization." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 10.6 (1991), pp. 761–769. URL: http://dblp.uni-trier.de/db/journals/tcad/tcad10.html#SutanthavibulSR91.

[2] N. Beldiceanu and E. Contejean. "Introducing global constraints in CHIP". In: *Mathematical and Computer Modelling* 20.12 (1994), pp. 97–123. ISSN: 0895-7177. DOI: https://doi.org/10.1016/0895-7177(94)90127-9. URL: https://www.sciencedirect.com/science/article/pii/0895717794901279.

[3] Helmut Simonis and Barry O'Sullivan. "Using Global Constraints for Rectangle Packing". In: (Jan. 2008).

[4] The Z3 Theorem Prover Project: Microsoft Research. *Z3*. URL: https://github.com/Z3Prover/z3.

[5] Monash University. *MiniZinc*. URL: https://www.minizinc.org/.